Method Writing Guide — httpd version

Introduction

This document is meant to be read when you have become confused as to what you need to do to write a **method** for the JGOFS data system. Then **method** is responsible for translating from the storage format/system of the data set into the transmission protocol(s) used by the data system. The **method** is required to

1) Accept the parameters — a composite of the arguments in the user's specification of the object (the arguments in parentheses) and those in the dictionary. Typically, the arguments consist of filenames, selections and projections. They are passed to the method as command line arguments.
2) Extract the information from the data set which satisfies the user's selections and projections.
3) Format the information appropriately (in the JGOFS protocols) and print it out.

Since some of these tasks are rather involved (especially parsing and interpreting selections such as `(station=4|station=5)&press<=500`, we have provided an outer shell, called "**outer**," which handles many of these functions and calls a set of subroutines provided by the method–writer. These latter (called "**inner**" in this document) are responsible for actually getting data from the dataset; they are the only part of the system which knows about the structure of the particular data set as it is stored on the disk.

The goal of this document is to provide you with the information useful in writing an **inner** routine and, with it, building, testing, and installing a **method**.

With the JGOFS server software, You will receive three examples of **inner** subroutines in the subdirectory **~jgofs/src**; they are called **def**, **lm**, and **nm**. See the document entitled "Default Methods" to see the data formats these deal with. In addition, there are detailed listings in the appendices. Also, the routine **outer** is in this subdirectory.

What actually needs to be done is to fit your data schema into our object–oriented approach. This isn't as difficult as it may sound; the basic set-up is very straight forward and this part of the documentation shows you how to do it. We describe the function of each subroutine within the collection of subroutines called **inner**. It will describe where the format information that you have about your data needs to go and some of the fancier things you can add to the basic set-up if you should desire. Remember, the important thing about this system is that one need not know what the **outer** programs are doing, only that each **inner** subroutine satisfies a need of the **outer** group of subroutines. Mastering how to satisfy these requests is accomplished by programming the **inner** routines.

What follows is a brief overview of each **inner** routine. While presenting this overview, some of the fancier things you can do are pointed out. Then the steps necessary to compile and link a new **method** are given. Then finally this primer finishes with an explanation of how to use a method to create an object, since you now know how the method was built, using the system should be less mysterious. In the Appendices are examples of a C and a FORTRAN method.

Inner *Overview*

The names of each of the routines in **inner** all begin with "*io*" (for the obvious reason). Below is a brief description of each subroutine as they appear in the C and FORTRAN versions.

---

ioopen: *integer function*, expected to open the data file at the top most level, determine the names of the variables, and their number.
ioopenf: *integer function*, the FORTRAN version.
iocommout: *integer function*, returns the next comment.
ioname: *subroutine*, returns the name of the v_n^{th} variable
iovarlevel: *integer function*, gives the level of a variable
ioattrout: *integer function*, puts out the next attribute of v_n^{th} variable
iovalreal: *subroutine*, returns value of v_n^{th} variable as a real number
iovalstr: *subroutine*, returns value as a string
ioreadrec: *integer function*, reads the next record of data at a particular level
ioclose: *subroutine*, closes all files

(In a C **inner**, the routine names are followed by an underscore _, *e.g.*, `ioname_`, for compatability with FORTRAN). These subroutines **must** be provided in your **method**, however, other subroutines may be added to take care of various housekeeping tasks within the **method** you provide.

Before entering into a more detailed description of each subroutine we will point out the things that they hold in common with each other. Generally, they will need to be able to share information through global variables or the method's own common block. You can call the common block by any name you prefer since the contents never need to be passed outside of the **inner** set of subroutines, but we have generally called it *method*. Often the character variables which hold the names of the variables and perhaps those holding string representations of the values may need to be common to several subroutines (since they may be ingested in *ioopen* and returned to **outer** by *ioname*, for example. Since the **outer** routine communicating with your **inner** is written in C, variable numbers start from zero, regardless of whether your routine is written in FORTRAN or C, all index counts start from zero. Additionally, all character strings should be terminated with a null character and there is a subroutine provided to help FORTRAN users with this. Each subroutine in **inner** has a very specific job to do and is relatively easy to code, though *ioopen* and *ioreadrec* can be tricky.

ioopen/ ioopenf

```
int ioopen_(s,nparams,ntotal)
   char *s[];
   int *nparams,*ntotal;
integer function ioopenf(s,nparams,ntotal)
  character*80 s(nparams)
  integer nparams,ntotal
```

As stated above the primary function of this subroutine (actually an integer function in FORTRAN) is to open the data set at the top-most level. The arguments that are passed to it as input are a character parameter array `s` (the first of which is often the data file or header file) and the number of parameters `nparams`, or, the number of elements in this character parameter array. When it's done *ioopen* returns the number of levels in the data set as a whole (count started at zero). In addition it is usually helpful for it to determine, for internal use, a variable that contains the total number of variables in the data set. This is returned to the calling program in the `ntotal` variable. Typically what else is done is to blankout, or clear, the first element of the character parameter array so that the outer programs no longer have to deal with it. If you use any of the other parts of the `s` array, you must also zero them out as well. Quite often lengthly comments statements are found at the beginning of data files or the first of many data files and one may want to read them here and save them in a work array so that they can be printed out by *iocommout* later (see the NABE **method** example *bloom.f* at the end of this document. While opening the top level file it is useful to make use of the *err=* capabilities to help while debugging the **method**. Finally, since this routine is typically run only once, it is a handy place to put those statements that need to be executed only once (*e.g.* a **data** statement defining the names of the variables).

iocommout

```
int iocommout_(str)
   char *str;
integer function iocommout(str)
  character*80 str
```

This routine extracts the next comment and puts it in the string `str`. The function returns with a value of one. If there are no more comments, the function returns a zero.

ioname

```
ioname_(vn,str)
   int *vn;
   char *str;
subroutine ioname(vn,str)
  integer vn
  character*80 str
```

This routine returns the actual name of the v_n^{th} variable. The index number of the variable is passed to the subroutine (count started at zero) in variable `vn` and *ioname* returns a character string `str` (null character terminated). Usually this is a very straightforward and short subroutine, but on the occasions where the data set does not use standard names for their variables, the method may wish to translate them. It is important to recognize that the interoperability of this DBMS system is much more awkward if we don't all use, at least for data being served out, the same names for the same variables. A table of suggested names is included in the appendices.

iovarlevel

```
int iovarlevel_(vn)
   int *vn;
integer function iovarlevel(vn)
  integer vn
```

The **outer** program needs to organize the variables according to their hierarchical level, and this integer function supplies it. The actual level of any variable and how it is determine depend on the physical and logical structure of the data set(s). It can be quite simple, but generally those variables that change the least often will be found in level zero (*e.g.* the cruise identification). The next least rapidly changing variables will be found in level one (*e.g.* the station latitudes and longitudes) and so forth and so on until one arrives at the most rapidly changing data (such as the bottle cast nutrients).

ioattrout

```
int ioattrout_(vn,str)
   int *vn;
   char *str;
integer function ioattrout(vn,str)
  integer vn
  character*80 str
```

The function *ioattrout* again takes the variable index number and returns the next attribute connected with that variable. These attributes are things like the units the measurements are given in, the method used to make the measurement, and key references to the analytical techniques used. Conventionally, we are using "`attribute=value`" for all attributes except the units which are simply returned as a descriptor (*e.g.*, gm/kg). The value of the function is one if an attribute is being returned, or zero if there are no more attributes.

iovalreal

```
iovalreal_(vn,f)
   int *vn;
   float *f;
subroutine iovalreal(vn,f)
  integer vn
  real f
```

This subroutine returns in `f` the real number value of the variable whose index number `vn` is passed to it. Depending on how the variables are stored internally this could be a simple matter of just sending the number back. Quite often, however, it is more convenient to store the variables as character strings internally and convert them to numbers when the need arises. Remember, before converting a character string to a real number, remove any leading blanks that may be present in the character string. If the string is text and cannot be converted, return -9999.0 as a value.

iovalstr

```
iovalstr_(vn,str)
   int *vn;
   char *str;
subroutine iovalstr(vn,f)
  integer vn
  character*80 str
```

This subroutine is similar to the one above, but returns the character string equivalent of a variable based on its index number. The above warning about leading blanks also applies here. Remember, FORTRAN programmers, character strings, such as the ones you are creating here, need to be null character terminated.

ioreadrec

```
int ioreadrec_(level)
   int *level;
integer function ioreadrec(level)
  integer level
```

This is the subroutine that actually does the reading of records from your data set(s), it is here that the "format" information goes. The value of the level to be read is the single argument passed to this integer function `level` and a one is returned if the read was successful or a zero if an eof was encountered at this level. If the names of additional data files need to be created from the inforamtion read at a higher level, this is where it is done. In addition to *iovalreal* this is another location where mathematical manipulations of the data is frequently done. A lot of things are happening at this step and this subroutine can be one of the longest. It must keep track of what level it is operating at and detect eof's as well as protect against opening/reading from files that do not exist.

It helps to keep in mind the calling sequence for this subroutine. If, for example, you have a data set with three levels (0, 1, 2) to be returned, the `level` argument will begin at 0. If successful, the subroutine will be entered again with argument 1, and then with 2. Repeated calls will be made to get the next record at level 2, until the function returns a 0 for end of file at this level. The subroutine will then be called again at level 1, and then again descend to level 2 calls. When the level 1 returns an eof, the next call will be for level 0 information, and the calls then walk back down the tree. An eof at level 0 indicates the end of the entire data set.

ioclose

```
ioclose_()
subroutine ioclose
```

This subroutine takes no arguments and is expected to close all open files properly.

In addition to **inner**

There are a few subroutines that are of great use for FORTRAN programs and can be provided.

---

strcat a subroutine to concatenate two strings (given as arguments) and return the product in the first argument string

istrlen an integer function that returns the length of the character argument (not including the trailing null), useful in dealing with count from zero environments

fixstr a subroutine that finds the first non-blank characters of the argument string and affixes a null character at the end, null terminates

unfixstr a subroutine that undoes the one above

Compilation and Usage

The general steps to creating a new method could go something like:

1) make a copy of an appropriate version of **inner**, renaming it to the name of the method you are about to write,
2) edit this file to put in the logical and physical format of your data set(s),
3) after saving this new file, compile and link the method with a command such as
   ```
   cc ~jgofs/src/outer.o yourmethod.c -lm -o yourmethod
   f77 ~jgofs/src/outer.o yourmethod.f -o yourmethod
   ```
4) You can test "yourmethod" by executing it directly, giving it the parameters which normally appear on the command line; *e.g.*:
   ```
   yourmethod datafilename
   ```
5) The generic syntax for using this method with JGOFS applications would be
   application `[-options] "`**method**`(parameters)"`

A final note

Optimization of any **method** you write *will* make the DBMS system run faster and seem more responsive. But a word of warning, rather than worry about making the code as efficient as possible, we find it is more productive to get something running and worry about the fine tuning later.